**HDH CQ2017/11 Project 1: Shell**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1712689  Huỳnh Ngọc Quân  Quanhuynh1310@gmail.com | 1712759  Phạm Minh Thắng  1712759@student.hcmus.edu.vn | 1712698  Võ Văn Quân  1712698@student.hcmus.edu.vn |

**Abstract**

The Shell is a program that takes commands from the keyboard and gives them to the operating system to perform. In the old days, it was the only user interface available on a Unix like system such as Linux. Nowadays, we have GUIs in addition to CLIs such as the shell. In this paper, we will implement simple shell with a few common **internal command**, **history**, **single redirection** and **single pipe**.

# Introduction

A shell interface gives the user a prompt, after which the next command is entered. Command will be parse into arguments include: command, flag… and execute. The shell interface program provides a history feature to allow a user execute the last comment by entering. It also support single redirecting input/output, communication via a single pipe, simple features of the shell interface.

## Requirements

* Unix
* Gcc
* C language
* POSIX library

# How to implement

## Creating the child process and executing the command in the child

We will use *fork* function to creating the child process. A process waits for its children to terminate or stop by calling *waitpid* or *wait* function included. This is functions in POSIX library, a C standard library for POSIX systems.

To execute command, we will parse input into arguments include: command, flag… and execute it. We will use *execvp* function

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

pid\_t fork(void);

// Call once, return twice

// Concurrent execution

// Duplicate but separate address spaces

// Shared files

pid\_t waitpid(pid\_t pid, int \*startup, int options);

pid\_t wait(int \*startus);

//Calling wait(&status) is equivalent to calling waitpid(-1, &status, 0).

int execvp(const char \*file, char \*const argv[]);

With background process (*&*). We use function *singnal.*

signal(SIGTTIN, SIG\_DFL); // Terminal input for background process

signal(SIGTTOU, SIG\_DFL); // Terminal output for background process

signal(SIGCHLD, SIG\_DFL); // Child stopped or terminated

## History feature

We will save all executed command in *log.txt*. When we need execute history command, the most recent command. We will prompt !! and after that the last command in *log.txt* will be executed.

In the future, I can improve shell interface with executing specified command in history by prompt ![number] with this *log.txt.*

## Redirecting Input/Output

Redirecting input or output use redirect operator “>” or “<”. The shell interface will not receive input or output from STDIN or STDOUT file descriptor, it receive from file, we will replace STDIN or STOUT with file descriptor by *dup2* function. So we will use *create* function for input and *open* function for output.

We will replace STDIN or STDOUT file descriptor by file descriptor of *create* or *open* function. After that restore STDIN and STDOUT file descriptor. These are functions that we will use.

int creat(char \*filename, mode\_t mode);

int open (const char\* Path, int flags [, int mode ]);

int dup2(int oldfd, int newfd);

## Allowing the parent and the child to communicate via pipe

Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In UNIX operating systems, pipes are useful for communication between related processes. In this project, we only implement single pipe.

Implementing pipe feature is more complicated. We need backup STDIN and STDOUT file descriptor before opening a pipe and restore them after closing the pipe to shell interface can continue without bug.

We will read from pipe and write from pipe. So, after backup STDIN and STDOUT file descriptor, we open pipe and use dup2 function introduced in 2.3 to replace STDIN and STDOUT with pipe read and pipe write. We read and write in separating processes.

It is helpful to understand that reading and writing via pipe can cause race condition. What happens if reading and writing in parallel and they shared memory. It is interesting that POSIX library solved for us this. But race condition can happen when we combine *fork* and *pipe* function. This is important.

int pipe(int fds[2]);

// fds[0] for read

// fds[1] for write

# Flow chart of shell interface

# 

# References

1. <http://man7.org/> for function description.

[2] <https://help.github.com/en/articles/basic-writing-and-formatting-syntax> for README.md